UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE GRADO

Detection of retransmissions in 10G Ethernet using GPUs

Paula Roquero Fuentes

Tutor: Iván González Martínez

June 2014

UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE GRADO

Detección de retransmisiones en 10G Ethernet mediante GPUs

Paula Roquero Fuentes

Tutor: Iván González Martínez

Junio 2014

Summary

Traffic analysis is an essential part of capacity planning, quality of service assurance and reinforcement of security in current telecommunication networks. As the network speed increases so does the traffic volume and the analysis of large traffic traces is computationally intensive.

This document presents a flow extraction software that allows obtaining TCP flow records at 4.4 Millions of packets per second in a single GPU directly from network. Such TCP flow records include number of retransmissions and duplicates per flow, which are very challenging to obtain at high-speed. Other records extracted are flag counters, zero window counters, duration and length of each TCP flow.

The software is designed as a pipeline where the packet data is analyzed and compacted in each stage to result in a brief record for each TCP flow after the last stage. Some stages of the algorithm take place in the CPU using *pthreads*, but the lion's share of the processing takes place in the GPU, freeing CPU cores for other uses. There is also a second program that helps with post-processing.

The high performance of the software comes at the price of precision, which is lost due to the implementation of the algorithm and limitations of memory. Our tests show that the loss of precision is small, so this tradeoff makes sense.

Our software significantly increases the processing power of the recently proposed high-speed sniffers based on commodity hardware and demonstrates the advantages of applying massively parallel processing devices for traffic analysis.

Resumen

El análisis de tráfico es esencial para planear la capacidad, calidad de servicio, garantía y refuerzo de seguridad en las redes de comunicaciones actuales. El aumento de la velocidad de red conlleva que aumente el volumen de tráfico, con lo que el análisis de trazas de red se hace cada vez más computacionalmente intensivo.

Este documento presenta un *software* de extracción de flujos capaz de obtener flujos TCP con una velocidad de 4.4 Millones de paquetes por segundo en una sola GPU que procesa directamente de red. Los flujos TCP contienen el número de retransmisiones y duplicados, que son difíciles de obtener a alta velocidad. Otros registros extraídos son contadores de *flags*, ventanas cero, duración y longitud de cada flujo TCP.

El *software* está diseñado como un *pipeline* en el que los datos de los paquetes son analizados y compactados en cada etapa para dar como resultado un breve registro de cada flujo TCP. Algunas etapas del algoritmo se ejecutan en la CPU mediante *pthreads*, pero la mayoría del procesamiento ocurre en la GPU, liberando núcleos de la CPU para otros usos. Un segundo programa ayuda con el post-procesado.

El alto rendimiento del software tiene como consecuencia una pérdida de precisión a causa de la implementación del algoritmo y limitaciones de memoria. Las pruebas muestran que la pérdida de precisión es pequeña, por lo que es un sacrificio aceptable.

El software mejora la capacidad de proceso de algunos *sniffers* de alto rendimiento propuestos recientemente y basados en hardware no especializado. También demuestra las ventajas de los dispositivos de procesamiento altamente paralelo para análisis de tráfico.

Keywords

CUDA, Duplicates, Flow register, GPGPU, GPU, HPC, Networking, Network analysis, Parallel programming, Retransmissions, TCP, Traffic analysis.

Palabras clave

Análisis de red, Análisis de tráfico, CUDA, Duplicados, GPGPU, GPU, HPC, Programación paralela, Redes, Registros flujo, Retransmisiones, TCP.

Index

Table index

Figure index

Glossary

- Berkeley Packet Filter (BPF): Language used to filter network packets.
- Capture file: File where the packets captured from a network are stored.
- Commodity hardware: Non-specialized hardware.
- CUDA: Parallel computing platform and programming model that runs in nVidia GPUs.
- CUDA kernel: Subroutine executed in the GPU.
- FPGA: Field-programmable gate array. Integrated circuit that can be reconfigured using a hardware description language.
- GPGPU: General-purpose Computing on Graphics Processing Units. The use of a GPU for computing not related to graphics.
- GPU: Graphics processing unit. Coprocessor used mainly to accelerate the creation of images. It also can be used for general computing (GPGPU).
- Mpps: Millions of packets per second.
- PCAP: Format of capture file.
- Pinned memory: Memory allocated in the host that cannot be swapped out to hard drive.
- RAW: Format of capture file.
- Sniffer: Device or software capable of intercepting and logging traffic passing over a network.
- TCP duplicate (switching): Packet that appears several times in a capture file or device due to the configuration of the capture device.
- TCP flow record: Record about a TCP connection with information such as IPs, ports, flags, etc.
- TCP ISN: Initial sequence number: Sequence numbers of the first SYN segments of a connection.
- TCP Keep-alive: TCP segment with the same sequence number as the last byte seen in the stream. It keeps the connection open without sending new data.
- TCP retransmission: Packet retransmitted by a sender after it remains unacknowledged for a timeout period.
- Wireshark: Program used to analyze capture files.

Introduction

An increasing number of services are being offered on top of IP, which is inherently best effort. This calls for network monitoring and traffic analysis, in order to ensure quality of service and perform capacity planning. Besides, traffic analysis plays a fundamental role in network security.

In the past, the traffic volumes were small enough to be managed with a sniffer device. Nowadays, the traffic volumes are huge and the sniffers have evolved to sophisticated systems that not only perform packet capture at line rate but also take care of storing and processing the captured packets. In this light, flow record extraction is a fundamental functionality, as it allows to inspect the traffic at the flow level, which a much smaller dataset than the packet trace itself. Then, once the time interval, hosts, protocols or ports of interest have been identified the corresponding packets from the trace can be extracted and further analyzed. Furthermore, there are *flow collectors* deployed in the network management centers that collect flow records from different network segments and perform correlation or issue alarms whenever an anomalous condition happens. The proposed GPU-enabled system can act as a flow record generator for such systems.

The most common flow record standards are Netflow (Claise, 2004) and IPFIX (G. Sadasivan, 2009). The flow record fields typically include the IP source and destination address (possibly with MAC addresses as well), flow size and duration and other parameters such as the number of TCP RST (reset) flags detected. Such parameters have different requirements in terms of processing. For example, the flow size can be easily calculated by adding up all the packet sizes in a flow, sequentially as they appear in the trace, even if they come out of order. However, when it comes to compare fields from many different packets in the trace the processing requirements are very stringent.

In this document, we restrict ourselves to TCP flow records, which are more challenging in terms of processing. Precisely, this is the case for duplicates and TCP retransmissions. The concept of TCP retransmissions is well known, whereby a TCP sender retransmits unacknowledged packets after a timeout period expires. In turn, a duplicate of a packet may appear in a trace because the same packet inbound to the VLAN is transmitted outbound of the VLAN eventually. If the whole VLAN is captured then a packet copy will be generated (For example by setting up a SPAN port of the whole VLAN). However, chances are that the packet is not a byte-per-byte duplicate but the same packet with TTL field decremented by one. This is case of traffic sniffing at both ends of an intermediate router, namely with a layer 3 hop in between. We call the former a "switching" duplicate and the latter a "routing" duplicate (Inaki Ucar, 2013). We only deal with switching duplicates in this document.

We note that the percentage of packets retransmitted per TCP connection is a relevant statistic because the more retransmissions the worse the quality of service, specially for bulk data transfers. On the other hand, it is extremely important to detect duplicates in the trace. If not, severe bias may be introduced in commonly used traffic statistics such as flow size and duration.

Note that the detection of both TCP retransmissions and duplicates are very demanding tasks in terms of processing, as many different packets must be compared to one another. Actually, packets may arrive out of order and the potential duplicate or retransmission may be located totally out-of-sequence. To complicate matters, a circular buffer is required that temporarily stores the packets (or the packet fields of interest) from a given connection in order to compare them, the larger the buffer the more the accuracy. Fortunately, such comparison task is well suited for parallelization, as every packet has to be compared with the neighbors in the same TCP connection.

On the other hand, the use of GPUs facilitates the adoption of ad-hoc hardware for traffic capture and analysis.

The research community has paid attention to the utilization of flexible and cost-aware solutions based on commodity hardware (L. Braun, 2010), in contrast to FPGA-based approaches (Florian Braun, 2002) and other commercial solutions (Cisco). The advantages of using commodity hardware are twofold. On the one hand, the amount of investment involved in the purchase of specialized hardware exceeds in several orders of magnitude the price of commodity hardware-based solutions. On the other hand, it provides more flexibility to adapt any network operation and management task as well as to make the network maintenance easier. As an example of this, we highlight the special interest that software routers have recently awakened (K. Argyraki, 2008) (M. Dobrescu, 2009) (S. Han K. J., 2010). Moreover, the utilization of commodity hardware presents other advantages such as using energy-saving policies already implemented in PCs, better availability of hardware component updates and flexibility in the implementation of novel measurement techniques.

Precisely, there have been many efforts to improve the packet capturing capability to 10 and 40 Gbps. However, the issue of how to process the traffic, namely how to extract the statistics of interest from the traffic trace, has not deserved the same attention. If the traffic volume is large, it turns out that the processing bottleneck is significant, specially for statistics that involve the comparison of fields from many different packets in the trace, as noted before.

We note that the packet capturing capabilities are normally based in Receive-Side Scaling techniques that basically divert the incoming traffic through separate hardware queues, which are subsequently handled by CPU cores in parallel (J.L. García-Dorado, 2013). By separating traffic in different queues, the throughput per queue decreases, which alleviates the load per core in the packet capture. As a result, as much as 14.7 Millions of Packets per Second (Mpps) can be captured, which is the case for a fully utilized 10 Gbps unidirectional link with small-size packets (64 bytes).

As attractive and cost-effective commodity hardware solutions may be, we note that *the fundamental limitation is in the number of cores*. In fact, most available

solutions (S. Han K. J., 2010) (N. Bonelli, 2012) (Rizzo, 2012) consume as much as 12 cores for a line rate of 10Gbps and typically 8 cores just for the packet capture, all of them with a very high utilization. Most importantly, we note *that this is the number of cores occupied per network interface*. Typically, there are several active interfaces per probe because several network segments have to be measured concurrently and some packet tracking between them may be performed. Therefore, there is little room for processing packets. Furthermore, not only the number of cores involved is important for the traffic processing but also the availability of memory and hard disk. Concerning memory, we note that the processing cores consume memory for packet capturing, because large buffers are needed to absorb the peaks. Concerning hard disk, we note that the drives are typically loaded due to packet storing at high-speed.

In this document, we focus on how to obtain TCP flow registers by means of massive parallel programming in GPUs, which is a packet processing task typically handled by CPU cores not devoted to packet capturing. We focus on the traffic parameters within a flow that are computationally hard to obtain, such as retransmissions and duplicates. The main advantage is that *the GPU increases the processing density of the commodity hardware*, namely it does not fully utilize additional cores. Furthermore, the throughput obtained is around 4.4 Mpps in a single GPU. This is sufficient for a real traffic scenario of 10 Gbps with an average packet size of 500 bytes, namely an approximate rate of 2 Mpps. On the other hand, the GPU also increases the memory density of the commodity hardware due to the internal memory, which can be used to absorb peaks at higher rates.

Our findings show that the GPU processing power matches that of 4 CPU cores system working in parallel in the best case for the CPU of perfect synchronization between cores. In a highly dense commodity hardware system such cores can be re-used for other packet or flow-record processing tasks, such as to run anomaly detection routines. Overall, the commodity system processing capabilities are greatly enhanced and the resulting GPU-enabled system becomes a real workstation for traffic processing at very high speed, beyond mere traffic capturing and storing.

The document is structured as follows. First, we introduce the state of the art and explain what we mean by duplicate and retransmission, in order to understand this important part of our parallel processing algorithmics. The parallel algorithm in-depth description, along with implementation details, follows. Then, we present the performance evaluation, both in terms of accuracy and throughput. Finally, we present the conclusions and future work. However, before we proceed with the technical agenda, let us briefly present some introductory material about GPUs, for the sake of completeness.

Introduction to GPUs

A GPU (Graphics Processing Unit) is a hardware for graphic rendering that can be found nowadays almost on every PC and also in some smartphones or tablets. Due to its massively parallel architecture, the GPUs can run trillions of instructions per second for both graphical and non-graphical applications. A GPU that is used for non-graphical applications is commonly known as GPGPU (General-Processing Graphics Processing Unit). The performance reached by GPGPUs makes this hardware amenable for High Performance Computing (HPC) clusters (Kindratenko, 2009). In fact, some supercomputer vendors have included GPGPUs inside their parallel computer blades. An example can be the SGI UV and the Cray XK7 supercomputer, which both include NVIDIA GPUs. GPUs have been also used in other research articles about traffic classification (Su., 2008) (Vasiliadis, 2008).

NVIDIA GPU Architecture

There are many different GPU architectures and models, NVIDIA and AMD being the most popular. Much research and testing have been performed to evaluate which technology gives a higher performance (Chen., 2011). Given that NVIDIA's CUDA language provides, in general, greater control than other GPU languages, we have opted to use NVIDIA and its CUDA programming technology.

Typically, GPU devices are external to the CPU. CPU and GPU connect and communicate through PCIe (Peripheral Component Interconnect Express), which entails that a memory copy from the host to the GPU has to be performed. This fact can make a GPU very inefficient if the data copy takes much time compared to the processing time. The NVIDIA's GPU architecture consists of a large number of SP cores (Streaming Processors), grouped into SMs (Streaming Multiprocessors). The SPs are small processors able to perform integer operations and simple-precision operations. The SM also contains double-floating point units, several registers, a level 1 cache and a shared memory. Each SM shares these resources among its SP cores. In a similar way, every SM shares a L2 cache and the global memory between the others SMs. In the NVIDIA's Fermi architecture we can find up to 16 SMs each with 32 SP cores (NVIDIA Corporation). In the newer Kepler architecture, it is possible to find up to 15 SMs each with 192 SP cores and 64 DPUs (double-precision units) (NVIDIA Corporation). Figure 1 shows an overall design of a NVIDIA GPU architecture.

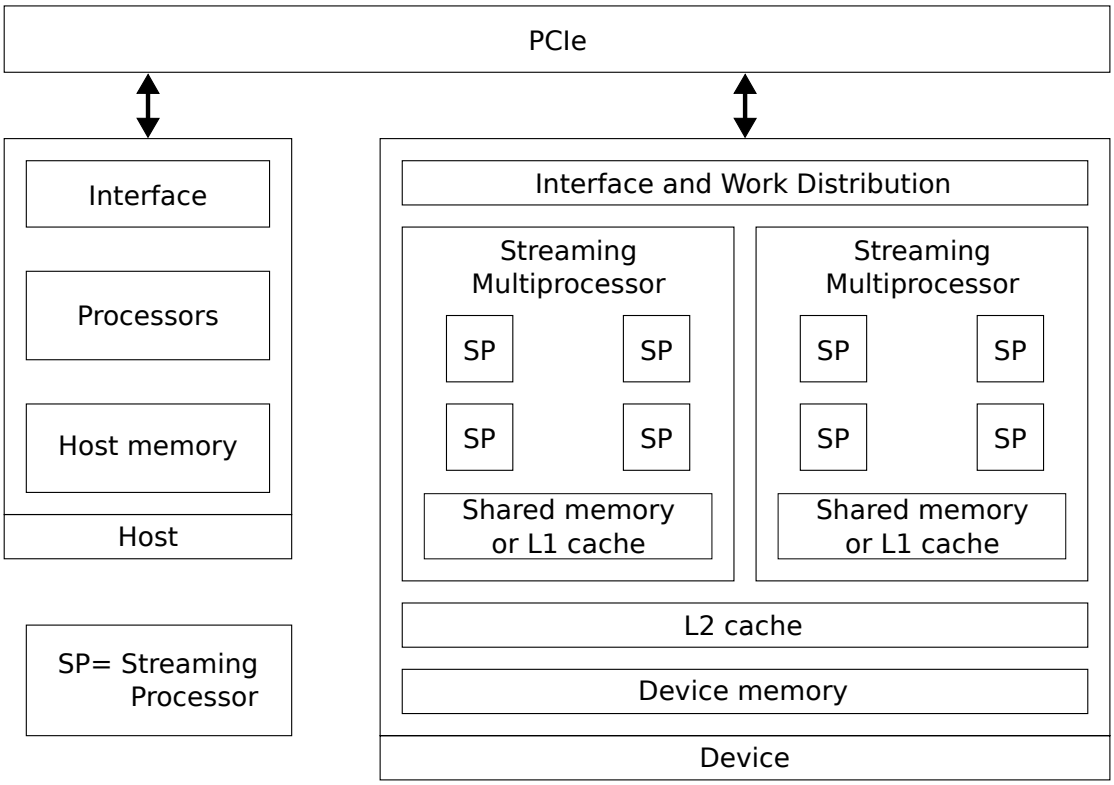


Figure 1: NVIDIA GPU General Architecture

CUDA Programing Model

The CUDA programing model enables to use parallel functions that are executed on the GPU, which are called CUDA kernels. Each kernel can be executed in parallel with other kernels if the device has the necessary resources available. Such kernel is launched on a grid, that is composed by a set of blocks (which can be defined as 1, 2 or 3-dimensional). In turn, each block is composed by a set of threads (that can be also defined as 1, 2 or 3-dimensional). In turn, each thread runs on a SP processor and each block is executed on a SM. Due to the architecture previously explained, different threads of the same block can share memory very efficiently (without having to access global memory).

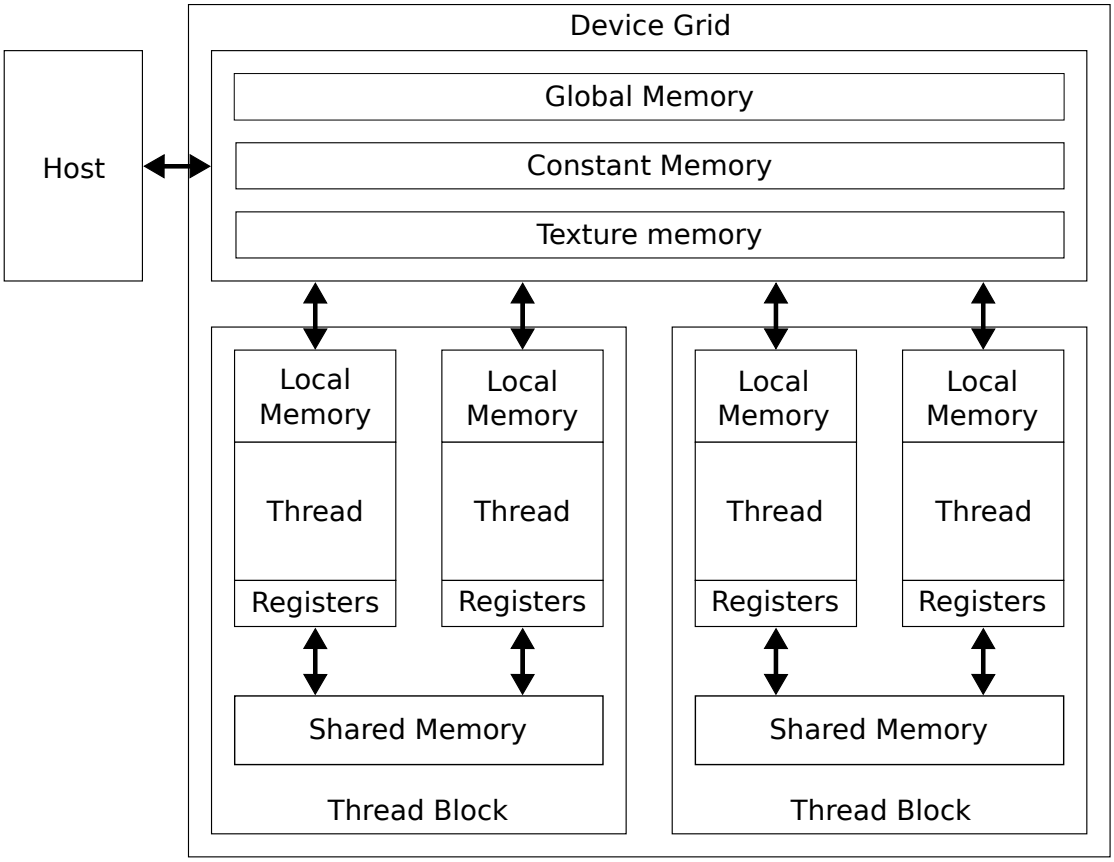


Figure 2: CUDA Programming Model

To obtain good performance, the programmer must ensure that the thread execution may not diverge in excess, as this would create serialization of execution between threads of the same block. The programmer must take into account the total number of threads and its distribution between blocks. Furthermore, the programmer should also consider the amount of shared memory used by each thread and other possible architectural considerations.

Figure 2 depicts how the CUDA programming model is organized. More information about CUDA programming model can be found in (Kirk).

State of the art

Once the GPU architecture and programming language has been presented we proceed with the state of the art. As it turns out, no previous work was found that deals with flow record extraction using GPUs, but there are a few references on processing traffic with GPUs.

Interestingly, we note that (S. Han K. J., 2010) investigates the use of GPUs in high-speed packet capturing and processing. The GPU is used to implement high-speed routing and pattern matching for anomaly detection, with excellent results. Even though the paper does not provide flow extraction (all the processing is performed at the packet level only) it demonstrates that the packets can be swiftly relayed from the CPU to the GPU at very high-speed (40 Gbps with small 64 bytes size packets).

On the other hand, Wu (Wenji Wu) shows how to use a GPU for packet filtering using the Berkeley Packet Filter (BPF). A performance comparison with a CPU was carried out and the CPU was actually faster. As it turns out, the data copy to the GPU does not pay off for the speedup achieved by the faster filtering.

Other authors have focused on flow record extraction using FPGAs (M. Zadnik, 2011) (S. Yusuf, 2008) (M. Forconesi, 2013). This is a completely different technology that allows to obtain an impressive line rate but at the expense of a much larger development time. Interestingly, none of the authors provides the TCP retransmissions and duplicates parameters in the flow record. This is because FPGAs lack memory space, which is essential to detect retransmissions and duplicates at very high-speed.

NetGPU is a framework designed to assist in traffic analysis using GPUs with CUDA. Both the design and implementation are described in a doctoral thesis (Clos, A framework for network traffic analysis using GPUs, 2010) and the code is available under free software license in Google Code (Clos, Netgpu).

The framework provides capabilities to read packets from several sources using PacketFeeders. Then, packets are distributed by an *analyzer* routine to several *analysis* processes, which are in charge of processing the packet in order to extract the desired statistics in the GPU. Then, the framework user implements the analysis routine to process the packets.

The framework defines buffers of fixed size for the packets. Some of these buffers can be joined before the data is sent to the GPU for processing. Actually, the documentation of the framework does not specify if there is a limit to the number of packets that can be accumulated. Even though this framework is interesting for packet processing it is not maintained at the present time and no performance figures are reported.

We employ hash tables to match packets with the corresponding retransmission or duplicate in the GPU. Let us briefly review the optimized hash library *cudpp* by Alcantara et al (Dan A. Alcantara, 2009) and discuss the limitations for our current work. Such hash table has a high performance and achieves the insertion of 5 million key-value pairs in 35.7 ms and access to all these pairs in 15.3 ms. To achieve this performance the hash table is implemented as a mix of sparse perfect hashing and *cuckoo* hashing, which makes use of the faster shared memory in the GPU to speed up its creation.

The high performance of this hash table implementation is mainly due to the use of 32-bit key and value, namely:

- Both the key and value can be written to memory in the same atomic access.
- Small-size keys and values allows to employ the fast shared memory, which is also very scarce, to speed up the creation of the table by means of *cuckoo* hashing.

However, we do require longer keys and values for the detection of retransmissions and duplicates. More specifically, the key must be 128 bits long and the value uses 192 bits, as will be discussed in the implementation section.

This prevents the adoption of the *cudpp* hash table. Furthermore, the variant of the algorithm described in the paper (Dan A. Alcantara, 2009) would not yield the desired performance increase. We note that the longer key and value impede the use of the shared memory, and this is key to obtain fast *cuckoo* hashing.

Retransmissions and duplicates

As noted in the previous section, the estimation of retransmissions and duplicates within a TCP flow is involved in terms of processing requirements. In this section we analyze how to estimate the count of retransmissions and duplicates in a flow. We note that there is tradeoff between speed and accuracy, namely it is not possible to detect all retransmissions and duplicates and keep line-rate operation at the same time.

Retransmissions

We consider that a TCP segment with data is a retransmission if the following TCP header fields are found in a previously arrived segment in the traffic stream: Source IP address, source TCP port, destination IP address, destination TCP port and TCP sequence number. In what follows, such array will be called a *quintuple*.

The above definition entails that only quintuples have to be compared to detect retransmissions, thus saving memory space in the GPU, which is scarce compared to the host memory. However, we do note that some retransmissions will not be detected (false negatives). Furthermore, some segments can be mistakenly detected as a retransmission (false positives). We will carefully review what are the false negative and false positive cases and come up with an algorithm that minimizes the occurrence of false positives. In the next section, we will perform a trace-driven analysis to quantify the overall accuracy of our algorithms.

A false negative may arise in the following cases. In what follows, we refer to TCP segments by their transmission order.

- When a TCP segment suffers segmentation at the TCP level, only the first fragment is detected as a retransmission because the other fragments have different sequence numbers. In Figure 3 segment 1 is divided into two segments, 2 and 3. Then, segment 2 is detected as a retransmission

but not segment 3 as it features a different sequence number that was not seen before in the traffic stream.

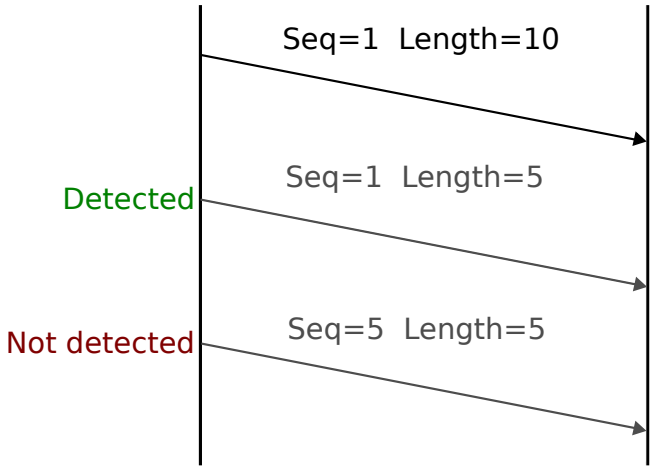


Figure 3: TCP segmentation

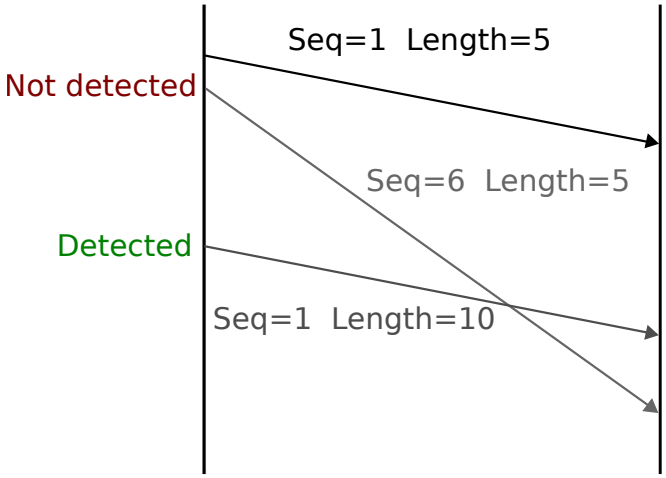


Figure 4: Joined segments

- When two or more segments are joined together only the first segment is considered a retransmission. In Figure 4 segments 1 and 2 are combined into segment 3, but segment 2 is not detected as a retransmission because its sequence number has not appeared before in the traffic stream.
- When TCP Keep-alive (SEG.SEQ = SND.NXT-1) segments are captured the first one is not considered a retransmission because its sequence number was not seen before in the traffic stream. An example is shown in Figure 5.

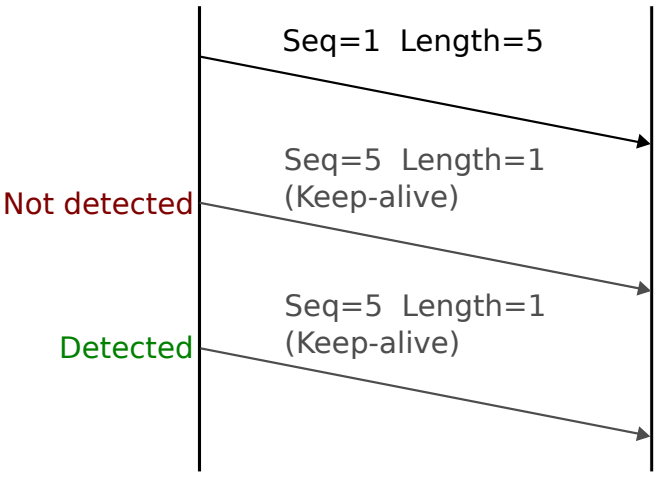


Figure 5: TCP Keep-alive

- Segments with common data. In Figure 6, a segment with SEQ = 11 and LEN = 5 was lost between segments 1 and 3. Afterwards, the lost segment was retransmitted (4) with more data, and it should be considered a retransmission because of the overlapped data. However, we cannot detect it because the sequence number was not seen before in the traffic stream.

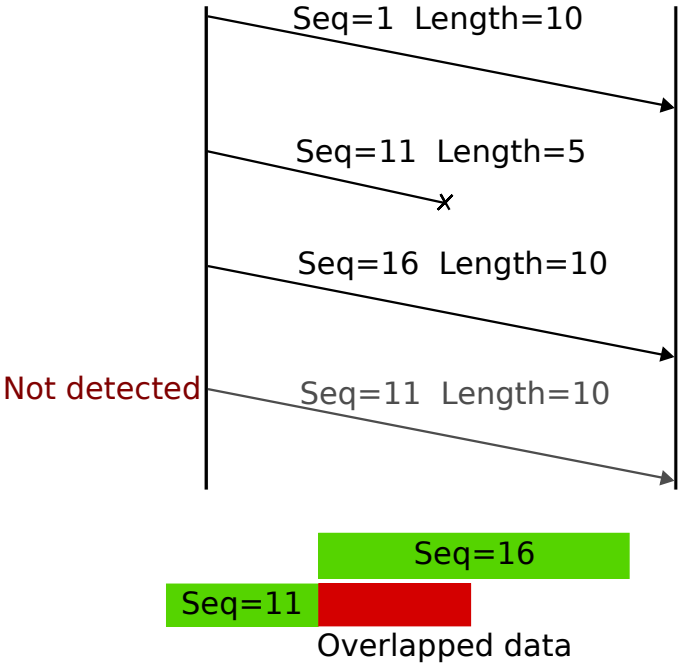


Figure 6: Common data

The latter cases constitute a small percentage of all the retransmissions, as will be analyzed in the performance evaluation section. On the other hand, chances

are that a non-retransmitted segment is tagged as retransmission, namely a false positive, if the connection is reused (*reincarnation*) and there are some segments with the same sequence number than others in the previous connection.

Even though this cases are not frequent, we further reduce the probability of a false positive by setting a maximum time between retransmissions, beyond which the potential retransmission will be discarded. As a result, our algorithm does not require knowledge of the ISN (Initial Sequence Number), which is normally used to detect reincarnations. In any case, we will provide a throughout performance evaluation later, that includes a quantification of false negatives and positives.

Duplicates

Concerning duplicates, we note that the quintuple must be equal between two segments, just like a retransmission, but additionally the IP packet identifier (IPID) must be equal. As explained in the introduction section, we only take into account the *switching* duplicates (Inaki Ucar, 2013).

Flow registers

For each flow, we obtain other parameters than the number of retransmissions and duplicates, such as a counter for the SYN, FIN and RST flags and also for the number of segments that announce a window size equal to zero but do not have the RST flag set. This is useful to detect congestion in the receiver side (receiver's window is exhausted). Note that the RST segments are not included in the counter because typically a RST segment announces zero window size to stop the transmitter on purpose. Namely, the RST flag does not indicate congestion at the receiver.

Lastly, the flow size in bytes and duration are obtained by means of the sequence numbers and timestamps of the SYN and FIN segments. This is a usual technique which have also been used elsewhere (A Papadogiannakis, 2013). If two different connections, possibly from a reincarnation, have the same source and destination IP address and TCP ports, then it is not possible to associate each segment to each particular connection. To prevent biased data we do not provide this counter if more than one SYN and FIN per connection are detected. This only happens in 0.84% of the connections.

Algorithmic design and implementation

This section describes the operation and implementation of the proposed algorithm to detect retransmissions and duplicates, count SYN, FIN, RST flags and zero window announcements and set the timestamp and sequence number of SYN and FIN segments. For simplicity, the single-CPU-thread version is described first. Then, the multi-CPU-thread version and the real-time packet capture and processing version will be presented.

Single CPU-thread implementation

The first stage of the algorithm performs the insertion of TCP segments into a hash table implemented inside the GPU using the quintuple as the key. The segments are read from the source file in blocks of size *0.7 * size of the hash table* to avoid increasing the number of collisions, and copied to the global memory of the GPU. After that, the kernel to find retransmissions and duplicates is launched in the GPU (*FindRetransmissionsKernel*). This kernel uses one warp per block (32 threads) and a block per TCP segment. The reason for this kernel configuration is to avoid deadlocks caused when part of the threads in a SIMT processor are locked in a branch that must be unlocked by the remaining threads in the other branch. This also has the benefit of doing coalescent accesses to memory. On the other hand, each block executes the following steps to insert a segment:

1. The segment is copied from global memory to shared memory.
2. A comparison is made between the current segment and the *n* previous segments stored in shared memory, in order to check if it is a duplicate.
3. The hash of the quintuple is calculated and used as a position in the table where the segment will be inserted (module size of the hash table).
4. If the bucket of the hash table is empty the segment is inserted.
    a. If the segment is tagged as a duplicate, the duplicate counter (dup_count) is set to 1.

b. Else the duplicate counter is set to 0 and the flag indicating that the bucket already has a non-duplicated (original_found) segment is set to 1.

5. If the bucket is in use, the quintuples are compared to check if the segment is a retransmission, provided that the time interval between them is less than the maximum time between retransmissions. If the segment is not a retransmission step 4 is executed in the next position of the hash table. Conversely, if the segment is indeed a retransmission, then
    a. If the segment is tagged as duplicate, the duplicate counter (dup_count) of the bucket is increased.
    b. If the segment is not tagged and a non-duplicate segment was already inserted (original_found == 1) the retransmission counter (retx_count) is increased.
    c. Else the bucket has a non-duplicated segment and original_found is set to 1.
6. The flag and zero window counters are updated. If the segment is a SYN or FIN the timestamp and sequence number are set.

Because the segments to analyze will fit into more than one memory block, it is necessary to design a mechanism to detect retransmissions and duplicates in two adjacent blocks. To detect retransmissions, once a new block is loaded into the hash table, only the segments from the next block with a quintuple already present in the hash table are inserted. Then, results are saved, the hash table is emptied and the remaining segments in the block are inserted normally. This process is shown in Figure 7. To detect duplicates, a buffer is used to save the last *n* segments of a block. Then the first *n* segments of the next block are compared to the segments in the buffer.

We note that such techniques eliminate border effects that happen whenever a duplicate or retransmission is present at the beginning of a memory block and the corresponding original packets are at the end of the previous block. However, it does not provide cure against false negatives that happen if the original is in the previous block but not at the end.

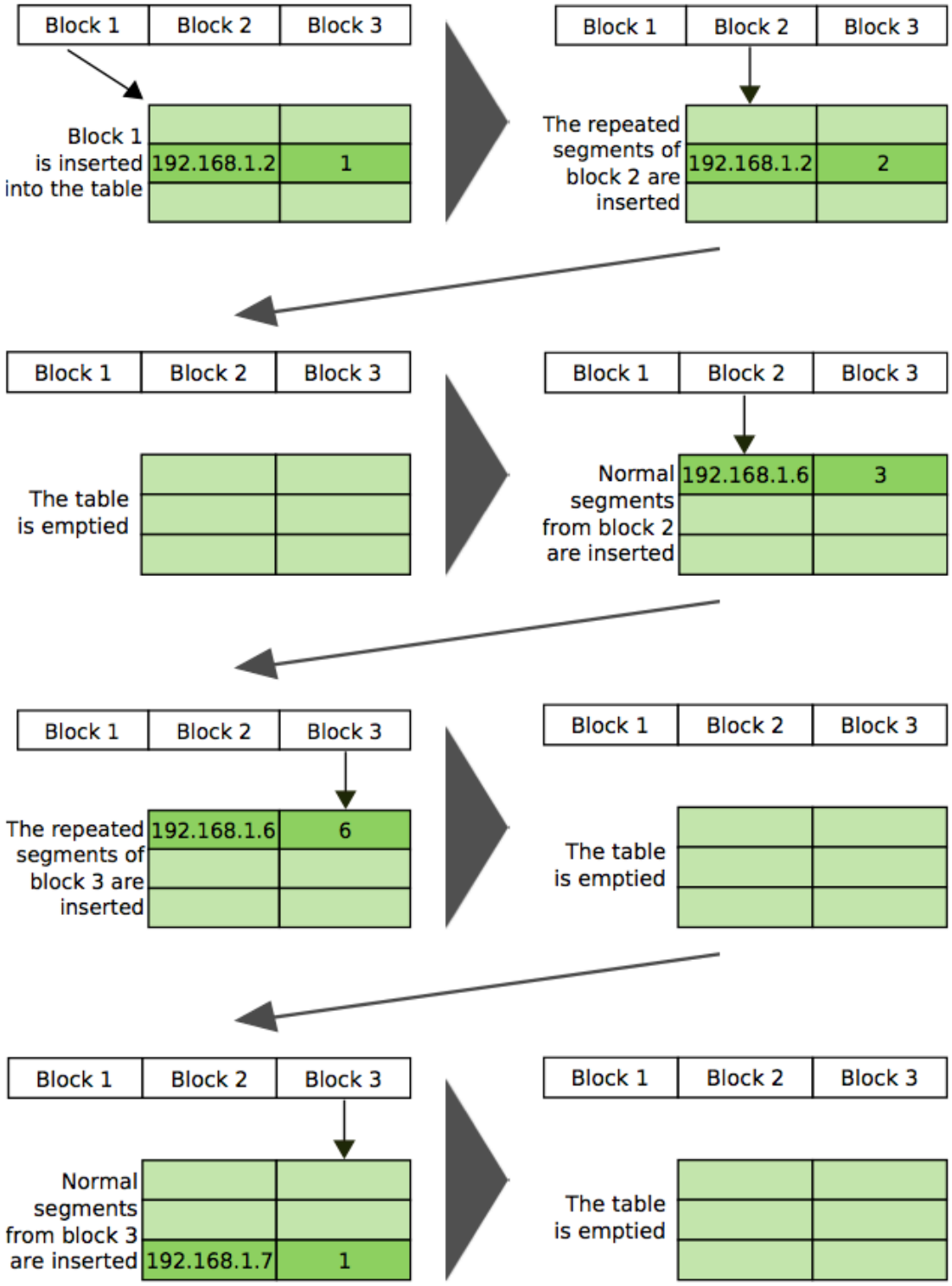


Figure 7: Insertion

A different hash table is used for connections, using source and destination IPs and ports as keys to the table. The counters of segments, retransmissions, duplicates, flags and zero window announcements are increased with each segment belonging to the connection. Such counters are stored in a single

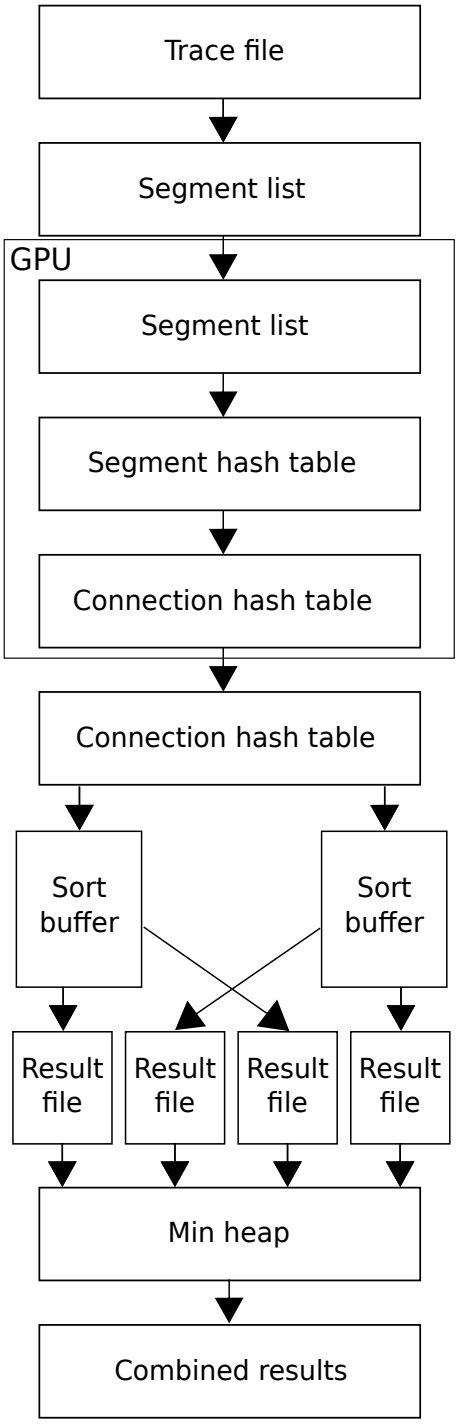


Figure 8: Data flow

int, using only one byte per counter. This is done to reduce the hash table bucket size, which allows to put more packet records into the GPU memory. In the performance evaluation section we will discuss that this is beneficial in terms of accuracy. To calculate the duration and length of each connection, the sequence numbers and timestamps of the SYN and FIN segments are stored. The FIN data size is also stored to calculate the total flow size. The drawback of this approach is that when several SYN or FIN flags are found for a connection, the length and duration cannot be calculated accurately, so the data is set as invalid, as noted before.

Another important design consideration is that the GPU memory is not large enough to combine all the segments from long-lived connections, so a post-processing is necessary to combine all the data from a connection. The connection data is copied from GPU to host, where the data is accumulated until a buffer is full. By using two different buffers we note that results can be copied in one buffer while the other buffer is being sorted.

Then, a priority queue is implemented to get the smallest quadruples from all the sorted files and combines the connection data when the quadruples are equal, adding the number of packets, retransmissions, duplicates and flow registers. This program uses the sequence numbers and timestamps of the SYN and FIN segments to calculate the length and duration of each connection. The final result

is a file with all the connection data sorted by quadruple starting with the smallest one. The whole process is shown in Figure 8.

Parallel implementation

Now that the basic algorithm has been explained as a serial process in the host, the real implementation using *pthreads* will be described. The use of *pthreads* increases the performance mainly because the GPU is calculating retransmissions while the host reads more packets from file. The program uses 4 threads synchronized with mutexes. Such threads read from file, insert segments, copy results to a buffer and write sorted results to disk. The threads are synchronized to protect the memory in each stage of the pipeline that appears in the Figure 8. Although there are 4 threads, normally only 1.5 are being executed. The stages of the pipeline are:

1. *Trace file or network*: Source of the packets.
2. *Host segment list*: Buffer in the host with the data of the relevant segments.
3. *GPU segment list*: Same buffer in the GPU.
4. *GPU segment hash table*: Hash table with retransmission data at the segment level.
5. *GPU connection hash table*: Hash table with retransmission data at the connection level.
6. *Host connection hash table*: Same buffer in the host.
7. *Host sort buffer*: Buffer where the connections are sorted.
8. *Result file*: Sorted connections.

Figure 9 shows the threads involved in the execution. The description follows, from left to right in the figure:

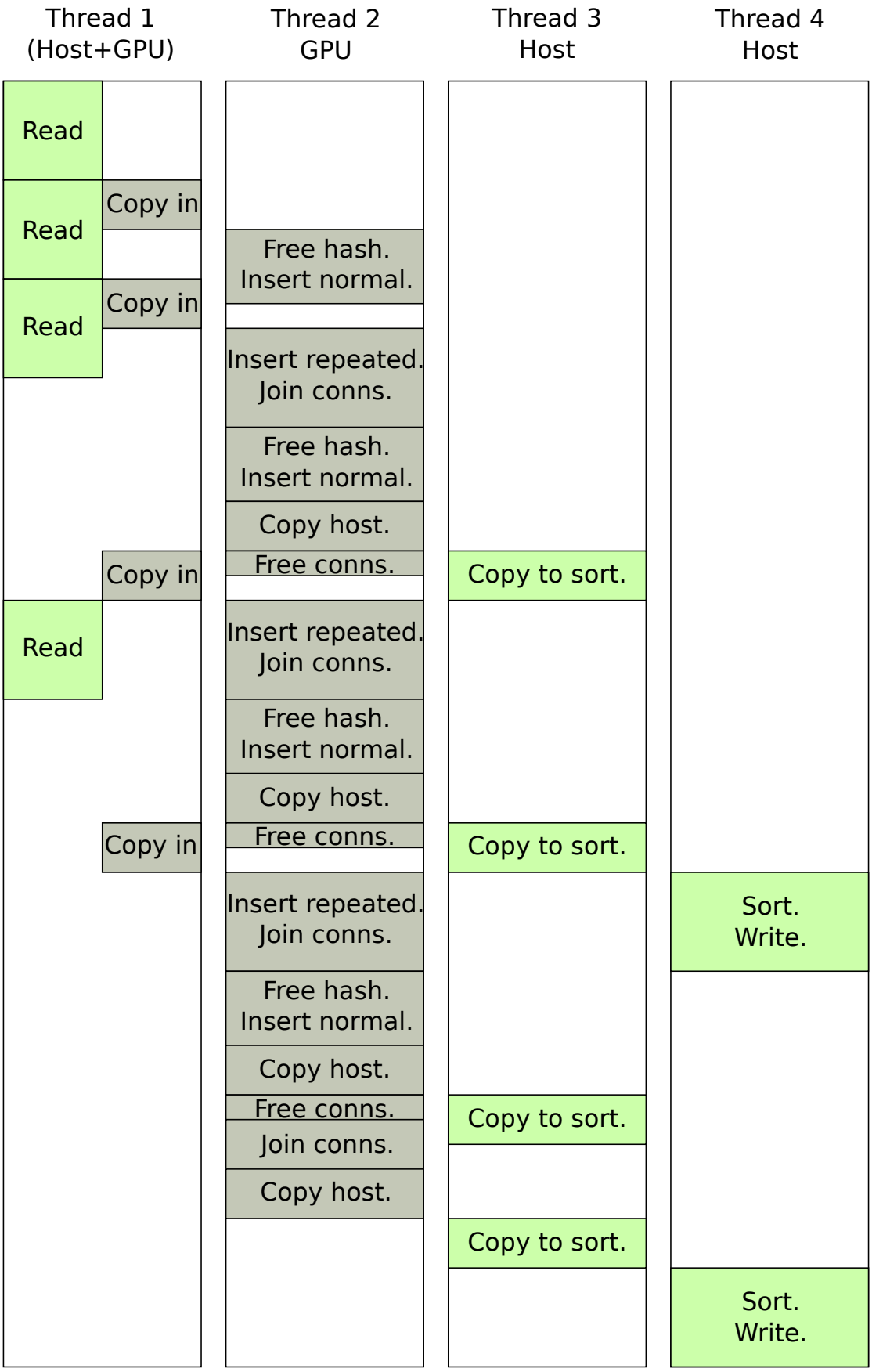


Figure 9: Time diagram showing *pthreads*

1. The first thread reads segments from *the trace file or network*, filters them and stores the necessary fields in *host segment list*. Once the GPU has processed the segments from the previous block, the segments are copied from host memory to the *GPU segment list*. The copy is asynchronous, so new segments can be read concurrently with the copy.
2. The second thread waits until the data in the *GPU segment list* is made available by the first thread. When the data is available the repeated segments are inserted to the *GPU segment hash table* and the data is joined for each connection in the *GPU connection hash table*. Once the segments in the *GPU segment hash table* are not longer needed, the table is emptied and the remaining segments in the *GPU segment list* are inserted. A mutex that allows the first thread to start copying new data is unlocked once the data in the *GPU segment list* is no longer needed. Before continuing, the thread waits in a mutex until the previous connection data in the *host connection hash table* has been copied to the *host sort buffer*. Then the data is copied from the *GPU connection hash table* to the *host connection hash table* and other mutex is unlocked so the third thread can start copying the current data to the *host sort buffer*. Finally, the *GPU connection hash table* is emptied.
3. The third thread waits in a mutex until there are results available in the *host connection hash table*, then copies the data to the *host sort buffer* and unlocks the mutex that allows copying more results from the GPU to the host. When the *host sort buffer* has enough data, a mutex is unlocked so the fourth thread can sort the results. Meanwhile this thread can copy the results to other *host sort buffer* so the sorting does not become a bottleneck.
4. The fourth thread waits in a mutex until the *host sort buffer* is full, sorts the results and writes them to the *result file*. Then another mutex is unlocked so the *host sort buffer* can be used again to store results.

Processing directly from network

To process packets directly from network no changes are necessary. The HPCAP custom driver, capable of capturing and storing traffic at 10 Gbps has been used (Moreno Martínez, 2012) for this purpose. This driver provides a block device interface that allows to send packets from an driver internal buffer to a regular file (/dev/hpcap). Traffic consumers can analyze or store the traffic at the same time and the speed of the system is that of the slowest consumer. Furthermore, the driver includes several internal buffers to shape traffic peaks for the consumers. However, after some tests to evaluate the performance, it was discovered that the driver's buffer was filled during the copy. This issue was due to the data copy to the GPU, that was synchronous (the CPU was blocked during the copy). To solve this problem, asynchronous functions to copy data to the GPU were implemented. As a result, packets can be copied to the GPU at any time while being received, without blocking.

Performance evaluation

First, we evaluate the accuracy of the algorithms, by taking several traffic traces as inputs and comparing results with a benchmark serial program that does not produce neither false positives nor negatives. The flows that showed discrepancy in the results were analyzed by means of *Wireshark* to find the reasons why. Once the accuracy was estimated we focused on the GPU throughput.

Datasets

With regard to datasets, the following three capture files have been used:

1. *duplicates.pcap* is a capture file with 100000 packets, out of which 99923 are TCP segments (53721 contain data). Moreover, 41426 packets are duplicates (switching) and only 5 packets are retransmissions. This dataset is intended for testing accuracy in detecting duplicates.
2. *retransmissions.pcap* is a capture file with 88423 packets, out of which 61990 are TCP segments (35360 contain data). Moreover, 1686 packets are retransmissions and there are no duplicates. This dataset is intended for testing accuracy in detecting retransmissions.
3. *retx_and_dup.pcap* is a capture file with 188423 packets, out of which 161913 are TCP segments (89081 contain data). Moreover, there are 41426 duplicates and 1691 retransmissions. This dataset is intended for testing accuracy in detecting retransmissions and duplicates and it is a merge of the previous ones.
4. *big.pcap* is a capture file with 200000000 packets, out of which 167409496 are TCP segments (109105217 contain data). This large dataset is intended to carry out random sampling of connections in order to determine the root cause of inaccuracies in retransmissions' detection, as will be explained later.

Regarding detection of duplicates, we note that neither false positives nor negatives were observed. We recall that a VLAN SPAN port produces duplicates,

which will be separated by the transit time through the router, that is typically very small. Then, original and duplicate packets are close together in the trace, thus falling either into the same memory block or in the border between subsequent blocks. As a result, duplicates can be easily detected by the GPU. The same does not apply to retransmissions, which are the focus of the next section.

Accuracy of retransmissions' detection

Table 1 shows the ratio of connections with true retransmissions not detected by the GPU for each of the dataset files (false negatives). Table 2 shows the same but with the number of retransmissions versus the total number of packets. We note that the connections without Initial Sequence Number (ISN) were not considered because the benchmark program requires the ISN to obtain the retransmissions. This is the case for connections with missing SYN packets, possibly due to a capture error because TCP connections require both SYNs from the client and server in order to be established.

Table 1: False negatives (connections)

	retx_and_dup.pcap	*retransmissions.pcap*	*duplicates.pcap*	*big.pcap*
Connections with retransmissions	531	528	3	304229
Connections with false negatives	73	73	0	41832
Percentage of false negatives	13.75%	13.83%	0%	13.75%

Table 2: False negatives (packets)

	retx_and_dup.pcap	retransmissions.pcap	duplicates.pcap	big.pcap
Number of retransmissions	1691	1686	5	1699364
Number of false negatives	84	84	0	87414
Percentage of false negatives	4.97%	4.98%	0%	5.14%

Conversely, Table 3 shows the ratio of connections with false retransmissions detected by the GPU and Table 4 shows the same but with the number of false retransmissions versus the total number of packets, for each of the datasets (false positives).

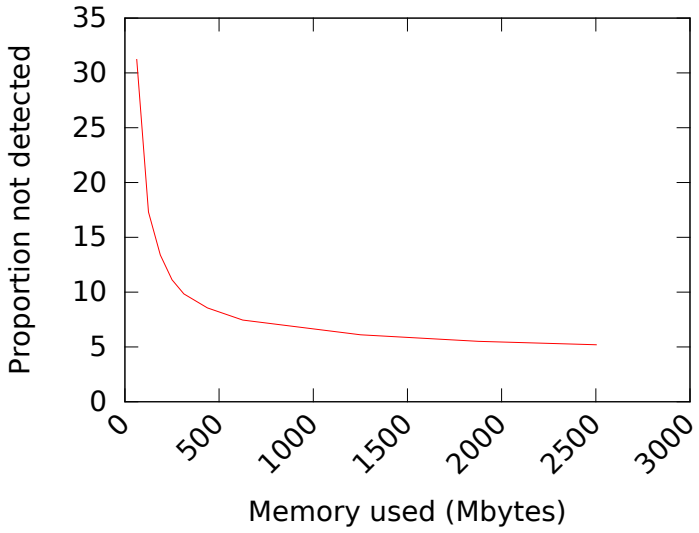
Table 3: False positives (connections)

	retx_and_dup.pcap	*retransmissions.pcap*	*duplicates.pcap*	*big.pcap*
Connections with retransmissions	531	528	3	304229
Connections with false positives	0	0	0	1
Percentage of false positives	0%	0%	0%	3e-4%

Table 4: False positives (packets)

	retx_and_dup.pcap	retransmissions.pcap	duplicates.pcap	big.pcap
Number of retransmissions	1691	1686	5	1699364
Number of false positives	0	0	0	20
Percentage of false positives	0%	0%	0%	0.001%

First, we note that duplicates are not mistakenly confused with retransmissions and the other way around, because the dataset with duplicates and retransmissions shows the same figures than the datasets with only retransmissions and duplicates respectively. Second, there are very few false positives and a significant ratio of false negatives. Consequently, we turn our attention to the evaluation of false negatives. We argue that retransmissions are not detected because they fall into different consecutive memory blocks sent to the GPU. Thus, the retransmission reaches the GPU after the original packet has already left. Figure 10 shows the ratio of retransmissions not detected (versus the total number of packets) for the *big.pcap* dataset versus the size of the memory chunk sent to the GPU.



The figure shows that the accuracy increases (conversely, the false negatives' ratio decreases) with the memory block size, reaching a lower bound of around 5%.

Figure 10: Ratio of undetected retransmissions versus memory block size

Why some retransmissions are not detected?

When the notion of retransmission was introduced we noted that some of the retransmissions could not be detected because of the detection algorithm adopted, that only takes into account the TCP segment quintuple. We denote such retransmissions by *structural* retransmissions, since they cannot be detected whatsoever. However, there are other retransmissions which are not structural and could not be detected either, the reason being the limited buffer size.

Recall that the packet trace is relayed to the GPU in memory blocks that fit into the GPU internal memory. If the original packet and retransmission do not fall within the same block then the retransmission cannot be detected. We denote such retransmissions by *split* retransmissions, as the connection they belong to is split into two different memory blocks.

Let $\mathcal{A}$ refer to the event that a given retransmission is not detected (i.e. a false negative) and let $\mathcal{A}_{struct}$ and $\mathcal{A}_{split}$ refer to the event that a given retransmission is not detected because it is either structural or split, respectively. Then,

$$\mathcal{A} = \mathcal{A}_{struct} \cup \mathcal{A}_{split}$$

$$\mathcal{A}_{struct} \cap \mathcal{A}_{split} \neq \theta$$

and we wish to have an estimation of $\mathbb{P}(\mathcal{A}_{struct})$ and $\mathbb{P}(\mathcal{A}_{split})$, noting again that structural retransmissions cannot be detected whatsoever. On the contrary, split retransmissions could be detected provided that more memory space in the GPU was available or, alternatively, that the traffic trace is demultiplexed -for example, by IP origin subnetwork- in different GPUs working in parallel. We also note that the $\mathbb{P}(\mathcal{A}_{struct} \cap \mathcal{A}_{split})$ is very small as it corresponds to the probability of a structural retransmission that is also split into two different memory blocks.

In order to isolate split retransmissions from structural retransmissions we performed a random sampling of connections in the *big.pcap dataset* as follows. Let *N* be the total number of flows in a dataset. Then, we randomly sampled a number of flows equal to *n such that the packets fit within a GPU memory buffer*. Note that this ensures that undetected retransmissions (false negatives) are *all due to structural retransmissions*. Therefore, such sample is valid to obtain the probability $\mathbb{P}(\mathcal{A}_{struct})$ by means of the estimate $\hat{p}$, which is equal to the ratio of retransmissions in the sample.

We note that the confidence intervals for such proportion are given by the Cochran approximation as follows:

$$\hat{p} \pm \lambda_\alpha \sqrt{\frac{\hat{p}(1-\hat{p})}{n-1}\frac{N-n}{N}} - \frac{1}{2n}$$

whereby λ_α is the corresponding percentile of the standard Gaussian distribution with significance level α.

A random sample of 100000 connections with 481000 retransmissions was used for the analysis. It was found that 4.31% (confidence interval [4.24, 4.38]) of the retransmissions were structural. Note that this is close to the 5.14% ratio obtained for the 4 GB memory block case and it shows that the probability of

split retransmissions has a lower bound of around 0.82%, and, as we deem $\mathbb{P}(\mathcal{A}_{struct} \cap \mathcal{A}_{split})$ small it must be close to $\mathbb{P}(\mathcal{A}_{split})$.

Furthermore, in a typical use case we are only interested in connections with a significant number of retransmissions (for example, connections with more than 5 retransmissions and 5% of the packets being retransmitted). In this case, only 0.31% of the connections have this problem. The statistical analysis shows that in 0.3% (confidence interval [0.25, 0.34]) of the connections the false negative was caused by structural retransmissions. Interestingly, as the number of retransmissions per connection grows so does the likelihood of falling into the same memory block, which dramatically decreases the probability of split retransmissions.

Accuracy of flow registers

In the last section, we have discussed that memory size is key to improve accuracy, because the larger the memory size the better chances to fit the original packet and retransmission in the same memory block. However, memory size is fixed in the GPU board and cannot be increased arbitrarily. That's why we follow the approach to compress the packet record and flow register in order to fit as many packets as possible in the memory block. Thus, a single integer is used to store four flow registers in order to reduce the hash table bucket size and increase the accuracy of retransmissions' detection.

However, we note that atomic operations, which must be used to update counters throughout the GPU code, only work with integers (32 bits) and some of our registers are 8 bit long, such as the counter of TCP flags per connection. As a result, we have to account for a possible overflow, which cannot be prevented. In this section we evaluate the impact in flow registers' accuracy. The file *big.pcap* was used to compare the differences in the number of flags and zero window announcements detected by the custom serial and GPU programs. The Table 5 shows the number of records showing differences in these two fields.

Table 5: Flow records showing differences in TCP flags and zero window announcements (serial vs GPU)

	SYN	FIN	RST	Zero window
Total records	54054	53971	53900	54103
Records with differences	27	19	122	22
Percentage	0.0005%	0.00035%	0.0026%	0.0004%

We note that the accuracy is remarkable in this case.

Throughput

An *nVidia Tesla C2075* was used to perform the throughput tests. A very large capture file of 1,4 Tb was used to minimize the effects of initialization in the execution time. This file had a total of 5.242.174.423 TCP packets and 81.290.811 connections with an average duration of 43,72 seconds.

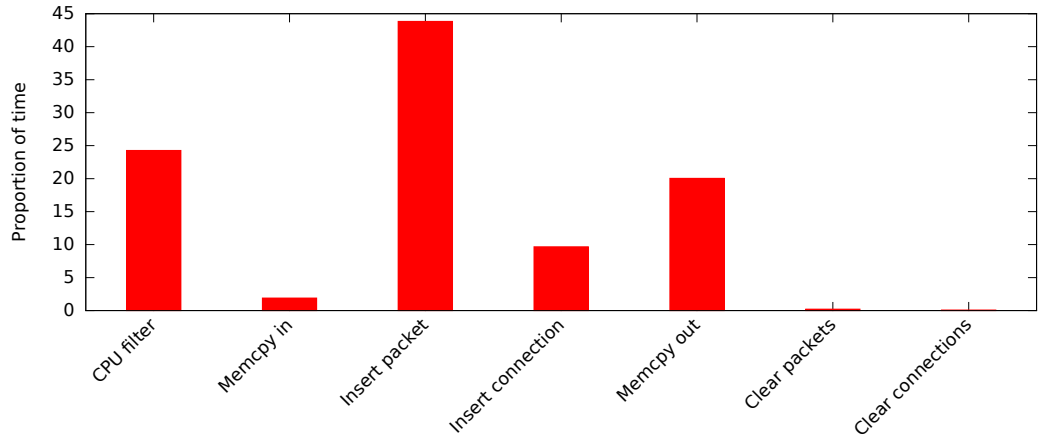


Figure 11: Proportion times

We note that the throughput may be bounded by the hard disk read speed and we actually get a 1.1 Mpps if we consider the whole disk and GPU system. This figure increases to 4.4 Mpps if we remove the disk read latency from the calculation. Figure 11 shows the percentage of time spent in the different execution stages. These stages are:

1. *CPU filter*: Filter the relevant packets.
2. *Memcpy in*: Copy the relevant packets from host to GPU.
3. *Insert packet*: Insert packets in the packet hash table.
4. *Insert connection*: Insert packets in the connection hash table.

5. *Memcpy out*: Copy connections from GPU to host.
6. *Clear packets*: Initialize packet hash table.
7. *Clear connections*: Initialize connection hash table.

We observe that the packet insertion task takes more time. Note that each packet has to be inserted into the packet hash table and, when using a GPU with 6 GB of memory, as many as 56 million of packets insertion are due. We note that the random access to memory is not well suited for a GPU. The "Packet to connection" task also involves a hash table, but the size is smaller because there are less connections than segments, so the performance penalty is not that significant. The time taken by the two *memcopy*s was variable even though pinned memory was used. The experiments were performed with a big pcap file (310 millions of packets), resulting in 12 executions of each part of the algorithm. This helps reduce the effect of the variability. The packet filtering takes a lot of time because it must be executed for each packet. The filtering phase takes place in the CPU because it was found that the GPU was slower and yielded a worse accuracy. Lastly, the two *clear*s empty the packet and connection hash tables for the next batch.

Comparison with CPU

To compare the performance, the GPU algorithms were rewritten to run in the CPU. The resulting program runs in a single thread and the number of threads necessary to achieve the same performance as the GPU is extrapolated from the run time in a single CPU core. This is the best-case scenario for the CPU solution, because the time lost in thread synchronization is not measured.

The GPU + CPU solution uses 1.5 processor cores and 3.7 Gb of memory in the host with the default options, while the CPU solution needs 4 processor cores and 7.16 Gb of memory to achieve the same performance and accuracy. As noted in the introduction, this is a significant savings in terms of cores, which is the bottleneck for traffic capturing and analysis at 10 Gbps in commodity hardware.

Capturing from network interface card

We have noted that the hard disk read speed is a limiting factor for the overall system throughput. However, the offline processing of a stored packet trace is not a real use case for the GPU, which will be typically working with live traffic from the 10 Gbps network interface card.

In this experiment, we actually assess that the measured throughput of 4.4 Mpps can be achieved when reading packets from the network interface card, i.e. a use case closer to the operational working environment. To do so, the same host used for the previous tests was connected to another host in charge of traffic generation (packet trace replay) at 10 Gbps through a 10 Gbps Ethernet link. The traffic traces were replayed at different speeds to measure packet loss and we achieved a 4 Mpps limit, which is consistent with the analysis in the previous sections. We verified that the throughput was limited by the GPU and not by the driver, which is able to capture traffic at 10 Gbps.

Conclusions and future work

In this document we have presented, for the first time ever, a GPU-based traffic capture and analysis system which is able to provide TCP flow records, including the challenging parameters of number of duplicates and retransmissions per flow. The fundamental breakthroughs to obtain a significant accuracy and speed have been studied, namely the limited memory size in the GPU board, which impedes to compare every packet with the previous ones in the trace. Despite of such constraints, the throughput for a single GPU system reaches a remarkable 4 Mpps figure. To put this figure into perspective we note that a fully saturated 10 Gbps link with 64-bytes packets produces 14.7 Mpps. However, a more realistic case of average packet size of 500 bytes, again in a fully saturated link, produces around 2 Mpps. We conclude that even though 10 Gbps line rate is not achieved the system is fast enough to cope with a typical 10 Gbps link in real operational conditions, which, in addition to the larger packet size, is not saturated.

However, as important as throughput may be, this is not the most salient advantage of the proposed system. As it turns out, the major constraint for traffic capturing and analysis at 10 Gbps in commodity hardware is the number of CPU cores available in the system. By carefully analyzing the state of the art we have found that the most commodity hardware systems rely on Receive Side Scaling (RSS) to demultiplex traffic at the network interface card into several queues, each of which being attached to a fully dedicated CPU core. The use of GPUs alleviates the load in terms of number of cores occupied in the traffic analysis. We also note that capturing the traffic only does not suffice for network monitoring: it is the analysis that matters.

As future work, we plan to achieve line rate by demultiplexing the incoming traffic stream into several GPUs, but not on a flow-per-flow basis, which demands a separate and possibly large hash table. Instead, we plan to use simple demultiplexing rules, based on the packet header structure.

Bibliography

Wenji Wu, P. D. (n.d.). *Network Traffic Monitoring and Analysis with GPUs*. Retrieved from http://sc13.supercomputing.org/sites/default/files/PostersArchive/tech_posters/post161s2-file3.pdf

Vasiliadis, G. a. (2008). Gnort: High Performance Network Intrusion Detection Using Graphics Processors. *Recent Advances in Intrusion Detection* .

A Papadogiannakis, M. P. (2013). Scap: stream-oriented network traffic capture and analysis for high-speed networks. *Proceedings of the 2013 conference on Internet measurement conference* .

Cisco. (n.d.). *Network Analysis Module (NAM) Products*. Retrieved from www.cisco.com/go/nam

Chen., Y. Z.-K. (2011). Architecture comparisons between Nvidia and ATI GPUs: Computation parallelism and data communications. *Workload Characterization (IISWC), 2011 IEEE International Symposium* .

Claise, B. (2004). *RFC 3954: Cisco Systems NetFlow Services Export Version 9.*

Clos, M. S. (2010). *A framework for network traffic analysis using GPUs.*

Clos, M. S. (n.d.). *Netgpu*. Retrieved from http://code.google.com/p/netgpu

Dan A. Alcantara, A. S. (2009). Real-time Parallel Hashing on the GPU. *ACM Transactions on Graphics* .

Florian Braun, J. L. (2002). Protocol Wrappers for Layered Network Packet Processing in Reconfigurable Hardware. *IEEE Micro* .

G. Sadasivan, N. B. (2009). *RFC5470: Architecture for IP Flow Information Export.*

Inaki Ucar, D. M. (2013). Duplicate detection methodology for IP network traffic analysis. *IEEE International Workshop on Measurements & Networking* .

J.L. García-Dorado, F. M. (2013). High-Performance Network Monitoring Systems Using Commodity Hardware. *Data Traffic Monitoring and Analysis: From measurement, classification and anomaly detection to Quality of experience.*

K. Argyraki, S. B.-G. (2008). Can software routers scale? *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow, Seattle, USA* .

Kindratenko, V. a.-M. (2009). GPU clusters for high-performance computing. *Cluster Computing and Workshops* .

Kirk, D. B.-m. *Programming Massively Parallel Processors: A Hands-on Approach.*

L. Braun, A. D. (2010). Comparing and improving current packet capturing solutions based on commodity hardware. *Proceedings of Internet Measurement Conference, Melbourne, Australia* .

N. Bonelli, A. D. (2012). On Multi–gigabit Packet Capturing with Multi–core Commodity Hardware.

NVIDIA Corporation. (n.d.). *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Retrieved from http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

NVIDIA Corporation. (n.d.). *NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110*. Retrieved from http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

M. Zadnik, L. P. (2011). FlowMon for Network Monitoring. *Networking Studies V: Selected Technical Reports* .

M. Dobrescu, N. E.-G. (2009). Routebricks. Exploiting parallelism to scale software routers. *Proceedings of ACM Symposium on Operating Systems Principles, Big Sky, USA* .

M. Forconesi, G. S.-B. (2013). Accurate and flexible flow-based monitoring for high-speed networks. *23rd International Conference on Field Programmable Logic and Applications (FPL)* .

Moreno Martínez, V. (2012). *Development and evaluation of a low-cost scalable architecture for network traffic capture and storage for 10Gbps networks.*

Su., L. W. (2008). Gregex: GPU Based High Speed Regular Expression Matching Engine.

S. Yusuf, W. L. (2008). Reconfigurable architecture for network flow analysis. *IEEE Transactions on VLSI Systems* .

S. Han, K. J. (2010). Building a single-box 100 Gbps software router. *Proceedings of IEEE Workshop on Local and Metropolitan Area Networks, New Jersey, USA* .

S. Han, K. J. (2010). PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review* .

Rizzo, L. (2012). Netmap: a novel framework for fast packet I/O. *Proceedings of USENIX Annual Technical Conference* .

Appendix A: Other configurations

The following tables show the precision of retransmission detection and the throughput of the program when different features are deactivated.

Table 6: Counting flags, zero window, duration and length of flows

	Throughput	% not detected
Default	4.4 Mpps	5.14%
Smaller connection hash table (connections dropped if they are small)	4.7 Mpps	5.03%
No insertion of repeated segments	5.6 Mpps	8.55%
No duplicate detection	5 Mpps	5.14%
All of the above	6.7 Mpps	7.9%

Table 7: No counting of flags, zero window, duration and length of flows

	Throughput	% not detected
Default	5.3 Mpps	4.5%
Smaller connection hash table (connections dropped if they are small)	5.6 Mpps	4.5%
No insertion of repeated segments	7 Mpps	6.9%
No duplicate detection	6.4 Mpps	4.5%
All of the above	8.2 Mpps	6.9%

Appendix B: Program help

First program (findRetx):

```
Usage:
findRetx -f input_file [-p format] [-u] [-o output] [-c hash_table_capacity] [-t max_time_between_retransmissions] [-d duplicate_window] [-r connection_proportion] [-s connections_per_outfile] [-e] [-i interface] [-a core] [-h]
	-f input_file : File with packets or list of files (with option -u).
	-p format : Format of files (pcap, raw) {default: pcap}
	-u : If present, the input file is a list of files.
	-o output : File where the result will be saved. Default is stdout
	-c hash_table_capacity : Number of buckets that will have the hash table in the GPU.
	-t max_time_between_retransmissions : Maximum number of seconds that have to be between two packets with the same quintuple to be considered retransmissions.
	-d duplicate_window Number of packets that will be checked to find a duplicate
	-s connections_per_outfile Approximate number of connections in each output file
	-r connection_proportion Expected proportion of connections to input packets. If this proportion is set lower than the true value some connections will be dropped.
	-e Don't insert repeated packets in previous block.
	-i Read using HPCAP from interface given.
	-a Core to use to read packets.
	-h Extended help of options and output format
Given a PCAP or RAW file, this program detects retransmissions and duplicates of the packets with data, joins the results for each connection and prints them.
This program writes a list of values with the following format:
1  srcIP Source ip
2  srcPort Source port
3  dstIP Destination ip
4  dstPort Destination port
5  totalPacketsSrcToDst Number of packets from src to dst
6  numberRetxSrcToDst Number of retransmissions from src to dst
```

7 numberDuplicatesSrcToDst Number of duplicates from src to dst
8 flagCounterSrcToDst Counter of FIN, SYN, RST and zero window
9 SYNValidSrcToDst 1 if the SYN data is valid from src to dst
10 SYNseqSrcToDst Sequence number of SYN from src to dst
11 SYNsecSrcToDst Timestamp of SYN in seconds from src to dst
12 SYNnsecSrcToDst Timestamp of SYN in nanoseconds from src to dst
13 FINValidSrcToDst 1 if the FIN data is valid from src to dst
14 FINseqSrcToDst Sequence number of FIN from src to dst
15 FINpayloadSrcToDst Payload size of FIN from src to dst
16 FINsecSrcToDst Timestamp of FIN in seconds from src to dst
17 FINnsecSrcToDst Timestamp of FIN in nanoseconds from src to dst
18 totalPacketsDstToSrc Number of packets from dst to src
19 numberRetxDstToSrc Number of retransmissions from dst to src
20 numberDuplicatesDstToSrc Number of duplicates from dst to src
21 flagCounterDstToSrc Number of FIN, SYN, RST and zero window
22 SYNValidDstToSrc 1 if the SYN data is valid from dst to src
23 SYNseqDstToSrc Sequence number of SYN from dst to src
24 SYNsecDstToSrc Timestamp of SYN in seconds from dst to src
25 SYNnsecDstToSrc Timestamp of SYN in nanoseconds from dst to src
26 FINValidDstToSrc 1 if the FIN data is valid from dst to src
27 FINseqDstToSrc Sequence number of FIN from dst to src
28 FINpayloadDstToSrc Payload size of FIN from dst to src
29 FINsecDstToSrc Timestamp of FIN in seconds from dst to src
30 FINnsecDstToSrc Timestamp of FIN in nanoseconds from dst to src

Second program (combineConnections):

```
Usage:
combineConnections input_file_start output_file
      input_file_start : Part of the name common to all input files. (For
files a_0 a_1 a_2 this would be "a_").
      output_file : File where the result will be saved.
Given the output of findRetx, this program combines the connection data.
This program writes a list of values with the following format:
1  srcIP Source ip
2  srcPort Source port
3  dstIP Destination ip
4  dstPort Destination port
5  totalPacketsSrcToDst Number of packets from src to dst
6  numberRetxSrcToDst Number of retransmissions from src to dst
7  numberDuplicatesSrcToDst Number of duplicates from src to dst
8  numberFINSrcToDst Number of FIN from src to dst
9  numberSYNSrcToDst Number of SYN from src to dst
10 numberRSTSrcToDst Number of RST from src to dst
11 numberZeroWindowsSrcToDst Number of zero window without RST flag from src
to dst
12 LengthSrcToDst Number of bytes from src to dst
13 secSrcToDst Time of the connection in seconds
14 nsecSrcToDst Time of the connection in nanoseconds
15 totalPacketsDstToSrc Number of packets from dst to src
16 numberRetxDstToSrc Number of retransmissions from dst to src
17 numberDuplicatesDstToSrc Number of duplicates from dst to src
18 numberFINDstToSrc Number of FIN from dst to src
19 numberSYNDstToSrc Number of SYN from dst to src
20 numberRSTDstToSrc Number of RST from dst to src
21 numberZeroWindowsDstToSrt Number of zero window without RST flag from dst
to src
22 LengthDstToSrc Number of bytes from dst to src
23 secDstToSrc Time of the connection in seconds
24 nsecDstToSrc Time of the connection in nanoseconds
```